C/C++ Contributing Editors

The New C: Integers, Part 2

Randy Meyers

The new C Standard has a novel idea: just accept that machine-word sizes will grow. Randy explains C's proactive strategy for accommodating the inevitable.

In December's column, I began discussing the new integer features of C99, in particular, the **long long** data type. In this month's column, I discuss how the C99 Standard generalizes the support for integers and allows your compiler to support additional integer data types. While it might not be immediately obvious, these C99 features were motivated in part by the introduction of 64-bit machines.

As the C99 Standard was being developed, 64-bit machines started to appear, and with them came the contentious issue of the mapping from C keywords to hardware integer data types. An informal group of vendors of 64-bit hardware and software met a few times to reach consensus on a common mapping, and failed to do so for a period of two or three years. Many of the proposed mappings called for a new integer type to be added to C. For example, one proposal mapped **int** and **long** to 32-bit integers and specified a new C type for 64-bit integers. Another proposal mapped **int** and **long** to 64-bit integers and specified a new C type for 32-bit integers.

Eventually, two 64-bit mappings won. Vendors most concerned about compatibility with their 32-bit offerings mapped **short** to 16 bits, **int** and **long** to 32 bits, and added a new 64-bit integer type (**long long**). Vendors most concerned about elegant access to 64-bit hardware mapped **short** to 16 bits, **int** to 32 bits, and **long** to 64 bits. (This is also the mapping used in Java.)

The C standards committee realized that there was a lesson to be learned from the 64-bit vendors: C probably would gain more integer types in the future. Someday, there will be interest in 128-bit integers. Occasionally, there is interest in adding unusual integer types, such as specialized counters for digital signal processors or integers with different "endian-ness" (byte ordering). The committee decided that it would be best if the Standard addressed the issue of adding new integer types to C in order to give direction to both implementations and programmers. (Some programmers complained about the addition of **long long** to the language since, rather than use typedefs or macros, they had hard-coded the information about the largest integer type in their programs.)

Generalized Integers

The model for *extended integer types* (as the Standard calls implementation-defined integers) should seem pretty natural to most C programmers. Like the *standard integer types* (as the Standard calls the integers with which you are familiar, such as **int**, **signed char**, and **unsigned long**), extended integers come in pairs of types. For every extended signed integer type, there is a corresponding extended unsigned integer type. Together the standard integer types and the extended integer types (along with **enum** types and **char**) are collectively known as the integer types. Thus,

all of the statements in the Standard about the integer types also apply to any extended integer types supported by an implementation.

The Standard does not say what the names of any extended integer types are. An implementation might name the types using a combination of current keywords (e.g., **long long long int**) or invent new keywords spelled using the name patterns that the Standard reserves for use by implementations (e.g., **__int24** or **_BigEndian32**).

Since C89, integers in C have been required to be binary numbers, and signed integers are required to be represented in one's complement, two's complement, or sign and magnitude notation. Integers (except **unsigned char**) may have unused bits in their representation. (On some machines, integers have the same storage representation as floating-point numbers with the exponent bits ignored.) Integers might have representations for both positive and negative zero. Extended integers obey these same rules. Thus, extended integers cannot be binary coded decimal. Since integers in C have fixed sizes, extended integers cannot be LISP-like **bignum**s whose storage dynamically grows and shrinks to hold numbers of unlimited size. Extended integers are like standard integers. If a programmer was given a typedef defined to be an integer, it would not make much difference whether the typedef was an extended integer or an unspecified standard integer.

Expression Evaluation

The rules for how extended and standard integers work in expressions is determined by a new concept called the *integer conversion rank.* The Standard requires every implementation to rank all of its integer types according to the following rules:

- No two signed integer types have the same rank even if they have the same representation. (That is, even if both **int** and **long** are 32-bit integers, one of them has greater rank than the other.)
- If one signed integer type is bigger (excluding any padding bits) than a second signed integer type, then the bigger type has greater rank.
- **long long** has greater rank than **long**, which has greater rank than **int**, which has greater rank than **short**, which has greater rank than **signed char**, which has greater rank than **_Bool**.
- An unsigned integer type and its corresponding signed integer type have the same rank.
- **char** has the same rank as **signed char** and **unsigned char**.
- An **enum** type has the same rank as the integer type used to represent it.
- If a standard integer type is the same size (excluding any padding bits) as an extended integer type, then the standard type has greater rank. (All things being more or less equal, give the standard type greater rank.)
- If two extended signed integer types are the same size (excluding any padding bits), their rank relative to each other is implementation defined, but still subject to the other rules for determining the integer conversion rank.

As you probably suspect, the integer conversion rank is used to define the rules for expression evaluation. Those rules are named the integer promotions and the usual arithmetic conversions.

The integer promotions rule states that an integer with rank less than **int** may be used in an expression anywhere where an **int** or **unsigned int** may be used (so can a bitfield). The integer value used in the expression is converted to **int** or **unsigned int**, depending upon which can hold all of the values of the original type. For example, if

short and **int** have the same representation, an **unsigned short** promotes to **unsigned int** when used in an expression.

The usual arithmetic conversions are the rules that determine the result type of most of the two-operand operators in C when they are operating on integer or floating-point operands. The usual arithmetic conversion rules state that first you perform the integer promotions on any integer operands. Then you determine a common type, convert the operands to the common type, and the result of the operator has that common type.

The first few rules for determining the common type for cases involving integers are pretty simple:

- If both operands have the same type, the common type is that type.
- If one operand has integer type and the other has floating-point type, the common type is the floating-point type.
- If both operands are signed integer types or both are unsigned integer types, the common type is the type with the greater rank.

Unfortunately, the Standard permits an implementation to make all of the integer types the same size, and that complicates the rules for when a signed integer type meets an unsigned integer type. Normally, if you multiplied a **long** by an **unsigned int**, you would expect the common type (and thus the result type) to be **long**. However, if **int** and **long** have the same representation, an **unsigned int** behaves like an **unsigned long** in terms of expression evaluation. Thus, when **int** and **long** are "the same," multiplying a **long** by an **unsigned int** is like multiplying a **long** by an **unsigned long**: the common type is **unsigned long**. The rules for when one operand has signed integer type and the other operand has unsigned integer type are:

- If the unsigned integer type has rank greater than or equal to the rank of the signed integer type, the common type is the unsigned type.
- Otherwise, if the signed integer type can represent all of the values of the unsigned type, the common type is the signed type.
- Otherwise, the common type is the unsigned type corresponding to the signed type.

While the integer conversion rank and expression evaluation rules may seem abstract, they are nothing more than the rules programmers assume about integers every day, particularly when dealing with typedefs. Consider a simple statement like: **x = y + z;** where **x**, **y**, and **z** all have the same typedef type and the only thing you know about that type is that it is a signed integer. You assume that the result of the addition has the same type as the variables, or that it is **int** if the type of the variables is smaller than **int**. You assume that the result does not overflow unless the true result would not fit in **x**. These are the sort of properties that result from the rules. They guarantee consistent and natural expression evaluation.

Constants

C99 does not introduce any new special syntax for integer constants that have an extended integer type. However, implementations might introduce such syntax as an extension. For example, if an implementation has a 128-bit integer type named **long long long**, it might allow you to suffix a decimal constant with **LLL** to indicate its type.

In C, integer constants have a type based on their value. For example, a decimal integer constant without a suffix has the first type from this list that can represent its value: **int**, **long**, or **long long**. The Standard permits a constant to have an extended integer type if its value cannot be represented by **long long** (or **unsigned long long**, if appropriate) and the extended type can represent its value.

The extended integer type used to represent a constant must be a signed type if the constant is normally signed (e.g., a decimal constant without a **U** suffix). The extended integer type must be unsigned if the **U** suffix appears in the constant. For octal or hexadecimal constants without a **U** suffix, the extended type may be signed or unsigned. (Unsuffixed octal and hexadecimal constants normally have the first type from this list that can represent their value: **int**, **unsigned int**, **long**, **unsigned long**, **long long**, or **unsigned long long**.)

Next Month

Next month's column wraps up integer support in C99 by discussing the new headers **<stdint.h>** and **<inttypes**.h>. These headers allow you to use both standard integer types and extended integer types in a more portable fashion.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*